Neuromorphic Computing: A Theoretical Framework for Time, Space, and Energy Scaling

James B Aimone
Neural Exploration & Research Laboratory
Center for Computing Research
Sandia National Laboratories
Albuquerque, NM 87185
jbaimon@sandia.gov

***Abstract*—Neuromorphic computing (NMC) is increasingly viewed as a low-power alternative to conventional von Neumann architectures such as central processing units (CPUs) and graphics processing units (GPUs), however the computational value proposition has been difficult to define precisely.**

Here, we explain how NMC should be seen as general-purpose and programmable even though it differs considerably from a conventional stored-program architecture. We show that the time and space scaling of NMC is equivalent to that of a theoretically infinite processor conventional system, however the energy scaling is significantly different. Specifically, the energy of conventional systems scales with *absolute* algorithm work, whereas the energy of neuromorphic systems scales with the *derivative* of algorithm state. The unique characteristics of NMC architectures make it well suited for different classes of algorithms than conventional multi-core systems like GPUs that have been optimized for dense numerical applications such as linear algebra. In contrast, the unique characteristics of NMC make it ideally suited for scalable and sparse algorithms whose activity is proportional to an objective function, such as iterative optimization and large-scale sampling (e.g., Monte Carlo).

I. Introduction

Neuromorphic computing ('NMC') hardware is increasingly available and can be implemented at near brain-like scales [1]. One implication of the success of implementing large-scale NMC systems is that it is increasingly evident that the biggest challenge facing the neuromorphic field, at least in the near-term, is the identification of which applications are well suited for its use. While scalable NMC platforms originally targeted large-scale brain simulations [2], [3], the growing need for low-power solutions at both the edge and in high-performance computing systems has increased interest in NMC to address energy challenges in computation broadly.

Key to their value proposition is that today's digital NMC systems are generically programmable—aside from some constraints of fan-in/fan-out, these systems can implement arbitrary computational graphs. Conceptually, this compatibility includes any algorithm that can be formulated as a threshold gate (TG) circuit, artificial neural network (ANNs), or any other more complex ensemble of neurons. As the ability to implement TGs confers Turing completeness [4], and ANNs confer universal function approximation [5], we can deduce that NMC is both universal in terms of algorithm compatibility and its ability to approximate functions.

Given this universality, the relevant question of NMC is not "*can* neuromorphic solve this task?", but rather "*should* NMC be used to solve this task?". While NMC is by definition general purpose in potential, based on the 'No Free Lunch' theorem as applied to computer architectures, it should be expected that it will be better at some tasks compared to others [6]. The value proposition of NMC is increasingly important given the increased use of other specialized architectures, such as general-purpose graphics processing units ('GP-GPUs', or simply 'GPUs').

This value proposition has been further complicated by the fact that the NMC field itself spans a number of different timescales and levels of technology readiness [7]–[9]. Unlike conventional architectures that are commercial today and emerging technologies such as quantum computing that will likely remain research platforms for the foreseeable future, neuromorphic research includes technologies that are near-ready for widespread adoption now as well as exploration of novel approaches that are many years out.

While there has been some increased attention to theoretical models for physical computing [10], there have been only a few efforts looking at the theoretical implications of scalable NMC platforms [11]–[13]. Despite the lack of a formal theoretical model, there has been considerable progress in recent years in identifying classes of problems that are well suited for addressing with neuromorphic algorithms. The goal of this analysis is to take a step back and look at how NMC architectures differ from Von Neumann architectures and systematically explore what classes of algorithms are well suited for neuromorphic computing.

This paper describes how NMC can be viewed as a class of specialized general-purpose architectures that is complementary to GPUs and other types of linear algebra accelerators that have become widespread in computing today. Figure 1 illustrates the notional hypothesis of this analysis: like GPUs, there exist several classes of computations that NMC excels at relative to conventional processors, and moreover that this is a distinct set of applications than what modern accelerators have targeted. Through the next few sections, we will discuss how NMC architecture differ from the conventional Von Neumann approach and we will show how these differences have direct impact on what types of computations NMC excels at relative

to more conventional architectural approaches.

II. Defining Neuromorphic Architectures in the Context of Von Neumann

NMC is typically described as a non-Von Neumann architecture, but rarely is it specified what type of architecture that it is. Here, we will discuss what the implications of being non-Von Neumann are and how this affects how we should view today's NMC platforms. However, first it is useful to briefly characterize what a Von Neumann architecture is and why it is so powerful.

A. Strengths and weaknesses of Von Neumann

At a very simple level, the Von Neumann architecture refers to a *stored-program* architecture wherein a memory external to the processor includes the instructions that represent a program as well as the data that will be processed. At the lowest level, the processor has specialized logic such as arithmetic logic units (ALU) that consist of a spectrum of specialized low-level circuitry to perform specific operations. From those basic operations, programs can be constructed to implement more sophisticated calculations. For example, a simple ALU has dedicated hardware to implement basic binary arithmetic (e.g., addition, subtraction), comparison operations (e.g., greater than, less than), and logical operations (e.g., bitwise logical AND, XOR). A Von Neumann program is simply a series of instructions that consist of which hardware-level operation to use, where in memory to pull inputs from, and where to store the outputs.

Von Neumann architectures are ubiquitous today for good reason. By separating processing and memory, each aspect can be designed and improved independently of the other. For instance, a powerful general-purpose processor, like a CPU, can dedicate considerable resources to having a wide range of increasingly sophisticated calculations hardware accelerated, while a more specialized processor can prioritize a subset of operations (as a GPU does for linear algebra), and a Reduced Instruction Set Computing (RISC) architecture may only have a lightweight set of instructions to maximize efficiency. Similarly, the separation of memory has allowed industry to focus on increasing density and access speeds, and with 64-bit addressing, there is effectively no limit to program or data size. This separation has also led to an important feature that has further entrenched the architecture—a serial program written fifty years ago in principle can run on today's hardware and vice versa. As a result, arguably this feature entrenched Von Neumann programs as the catalyst of the first Hardware Lottery [14].

The downside of Von Neumann is that scaling to larger systems with more memory and more powerful compute elements physically results in moving computation further away from memory. Stated differently, a bigger memory or a bigger processor requires that information—both data and the program itself—be moved longer distances. For this reason, the vast majority of the energy cost of computers today is in memory accesses [15], and much of modern computer architecture research focuses on this challenge [16]. This is one reason why Moore's Law was so critical, so long as transistor sizes were getting smaller, more memory could easily be placed nearer to the processing which effectively bounded the time and energy costs of having to go off chip.

B. Formal model definitions for neuromorphic computing

While the NMC landscape continues to evolve, it is useful for the remainder of this analysis to have a concrete theoretical model to refer to. Here, we define a concrete model of NMC and in the following sections we will analyze the distinctions of this model from Von Neumann and the implications of its hardware realization.

1) Neurons and Synapses: First, we provide generic definitions of the two key classes of primitives in a NMC system, *neurons* and *synapses*.

Definition II.1 (Neuron)**.** A **neuron** is a compute element that has at least one time-evolving internal state $x(t)$ that is partly updated by combining information from multiple inputs (i.e., 'fan-in' from other neurons) and a single observable state $y(t)$ that can be observed by other neurons. The neuron's computation is characterized by a non-linear transfer function, $f(.)$, such that $y(t) = f(x(t))$, and $x(t)$ evolves according to some dynamic, such as $\frac{dx(t)}{dt} = g(x(t), \mathbf{s}(t))$, where $\mathbf{s}(t)$ describes the set of inputs for the neuron.

This definition of neuron is generic and inclusive of any artificial neuron type. For instance, for TGs or McCulloch–Pitts neurons $f(.)$ is a Heaviside function; for modern ANNs $f(.)$ is often a sigmoid or rectified linear function.

Definition II.2 (Synapse)**.** A **synapse** is a compute element that represents the transfer of information from a source neuron, i, to a target neuron j. The computation of a synapse, $s_{i,j}(t)$ is typically defined as a function of at least the input neuron $y_i(t)$ and a synaptic weight $w_{i,j}(t)$, such as $s_{i,j}(t) = h(y_i(t), w(t))$. More complex synapses may include information of the post-synaptic neuron $s_{i,j}(t) = h(y_i(t), x_j(t), w_{ij}(t))$. Learning represents the change of the synaptic weight over time, as in $\frac{dw_{ij}(t)}{dt} = h_{learn}(w_{ij}(t), y_i(t), x_j(t))$.

This definition of a synapse similarly is generic and inclusive of any artificial synapse model. For standard ANNs and TGs, after training $w(t)$ is a constant and $h(.)$ is a linear function, such that $h(y_i(t), w(t)) = w \times y_i(t)$, allowing linear algebra methods to be used. Table I shows common definitions for the governing functions $f(.)$, $g(.)$, and $h(.)$.

2) Neuromorphic algorithms and architecture: As our goal is to construct an NMC system that is tailored for efficient neural computation yet remains programmable, we propose here that a generic NMC architecture is one which restricts the computation of neurons and synapses to a targeted set of functions.

Definition II.3 (Neuromorphic Computing Architecture)**.** A **neuromorphic computing architecture** is a distributed computing architecture that consists of specialized circuitry to

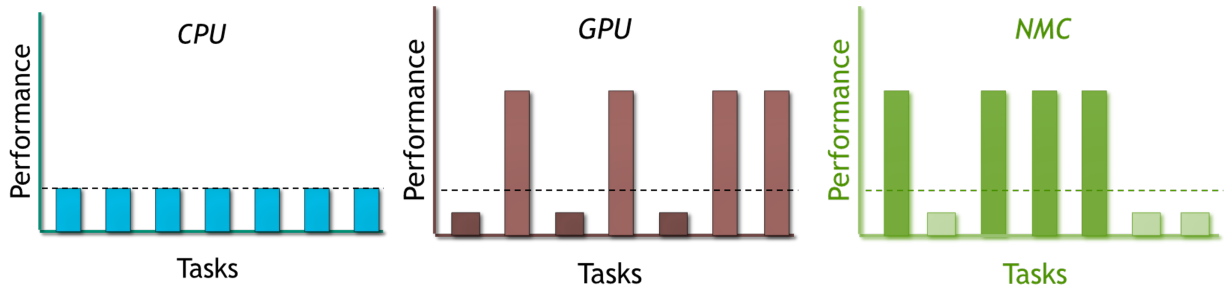

Fig. 1. NMC hardware is GPU-like in generality, but shows advantageous capabilities in a different set of tasks. This paper explores the classes of computation for which NMC shows preferential advantages

directly compute a class of governing functions for *neurons* and *synapses*. All required memory for parameterization of neurons and synapses is co-located with the circuitry required for their computation.

Even though their governing functions can vary, for any set of governing function we can define a neural algorithm as a graph of neurons and synapses.

Definition II.4 (Neuromorphic Algorithm)**.** If $G = (\mathcal{V}, \mathcal{E})$ is the computational graph describing a conventional algorithm, where each $v \in \mathcal{V}$ is an operation of G, then for a neuromorphic architecture, a **neuromorphic algorithm** is an algorithm that can be defined as the computational graph, $G_N = (N, S)$, where N is a set of *neurons* and S is the set of *synapses* that describe the *directed* connections between pairs of neurons that is isomorphic computationally to G. Let T be the total runtime of G. A subset of neurons $N_{in} \in N$ define the inputs to the algorithm provided at times $t_{in} \subseteq [0, T]$, and a subset of neurons $N_{out} \in N$ define the outputs of the algorithm observed at times $t_{out} \subseteq [0, T]$.

Neuromorphic algorithms can be defined to describe the neural graph necessary to implement either full-scale applications or small-scale assemblies of neurons that implement relatively simple mathematical operations, such as arithmetic functions [17]. Of note, in this paper the term 'circuit', when referring to neurons, refers to such graphs of neurons and synapses that target specific computational operations as opposed to 'circuitry', which refers to the physical hardware.

The Turing completeness and universality of neural circuits is for the most part independent of the governing functions [1]. Therefore, for any desired target function, for different governing functions there will be different neural algorithms. Understanding this neural algorithm equivalency is critical for mapping an NMC algorithm onto the governing functions available on a given architecture, and likewise is necessary for defining the computational trade-offs for evaluating the advantages or disadvantages of using an NMC-compatible algorithm. Stated differently, the equivalent leaky integrate-and-fire (i.e., 'spiking') algorithm for an ANN model will be a different computational graph that may require more or fewer neurons and synapses.

[1]For example, universal function approximation is proven so long as $f(.)$ is continuous, bounded, and non-linear [18].

C. Ideal versus realizable neuromorphic architectures

Definition II.3 defines an NMC architecture in a generic manner, however we can further distinguish an ideal NMC architecture from those that are practically realizable. This is useful because while the ideal NMC architecture is a useful strawman of sorts, we can explore the computational advantages of an ideal NMC approach and then back off of those with the costs (and benefits) of backing away from this ideal architecture to a practical approach.

Our definition of an ideal NMC architecture relates to the parallel structure of the architecture itself. Like the brain, every neuron in an ideal NMC system would be physically distinct from every other neuron and every synaptic connection would similarly exist physically distinct from all others (i.e., point to point connectivity). In contrast, today's NMC architectures leverage a conventional-like hierarchy of processing elements and network routing. For example, Intel Loihi cores have local memory that stores the state variables of many neurons and their associated synapses and leverage specialized circuitry to rapidly update those neurons' states. In effect, below the core level the architecture itself is not neuromorphic as much as it is a highly-specialized RISC stored program architecture. From an algorithm perspective it does not strictly matter that at the lowest level an NMC architecture has cores that are shared by many neurons; but this distinction from the ideal architecture does introduce notable savings (e.g., space) and costs (e.g., time, embedding constraints) that we must eventually consider.

What we do not include in the definition of an ideal architecture are the governing functions themselves (e.g., the activation functions of neurons or the computation within synapses). In part we do this to separate the potential functionality of the components from the architecture itself. This is rather fundamental as the brain is not programmable in the same sense as engineered hardware—the circuit is what it is, the functions are what they are. However, there is a more fundamental reason for dissociating the two. Modifications to the governing functions require a change the algorithms

TABLE I
NEURON AND SYNAPSE FUNCTIONS FOR DIFFERENT NEURAL COMPUTING FRAMEWORKS

	$f(.)$	$g(.)$	$h(.)$
Threshold Gate	$y_j = \begin{cases} 1 & \text{if } x_j > \text{threshold} \\ 0 & \text{otherwise} \end{cases}$	$x_j = \sum_i (s_{ij})$	$s_{ij} = w_{ij} y_i$
Artificial Neural Network	$y_j = \begin{cases} x_j & \text{if } x_j > 0 \\ 0 & \text{otherwise} \end{cases}$ or $y_j = tanh(x_j)$	$x_j = \sum_i (s_{ij})$	$s_{ij} = w_{ij} y_i$
Leaky integrate-and-fire	$y_j(t) = \begin{cases} 1 & \text{if } x_j(t) > v_{thresh} \\ 0 & \text{otherwise} \end{cases}$	$x_j(t) = \begin{cases} v_{reset} & \text{if } x_j(t) > v_{thresh} \\ x_j(t) e^{-dt/\tau} & \text{otherwise} \end{cases}$	$s_{ij}(t) = w_{ij} y_i (t - d)$

themselves [2]; whereas the ideal versus practical distinction here should not impact a neural algorithm inasmuch as it relates to the embedding and performance of that algorithm.

For the purposes of this analysis; it is useful to remember that 'neuromorphic' and 'von Neumann' represent ends of a broad spectrum of architectures, and today's NMC platforms sit between these extremes. That said, unless stated otherwise, for the remainder of this analysis we will explore NMC specifically in the context of an ideal NMC architecture.

III. THE BENEFITS AND COSTS OF MOVING TO NEUROMORPHIC FROM VON NEUMANN

Even though NMC is not Von Neumann, if it is to be used in a general-purpose manner, it ultimately must satisfy many of the same requirements, such as programmability, scaling, and reliability. As such, it has a distinct set of tradeoffs it must deal with. One-by-one, we walk through some of the positive and negative implications of NMC being non-Von Neumann.

*A. Neuromorphic computing is programmable, but it is **not** a stored-program architecture*

Once programmed, NMC systems generally do not have an external memory from which it serially retrieves the next step of a program continuously during operation. Rather, the program, as such, is directly encoded in the construction of the neural circuit on which information will be computed, a fact which both confers benefits but introduces some direct and indirect challenges.

The most immediate implication of this is that while NMC is programmable, it is initially best to consider that it has a fixed program for the duration of its operation. The program being fixed means that the whole program must be spatially deployed across the neural hardware. This physical realization of a model does have very real implications on what NMC can be used for. Generally, on today's scalable NMC platforms [1] the program is defined as the computational graph of neurons and synapses that directly represents the algorithm being implemented [19], as in definition II.4.

This explicit representation of a whole algorithm or application in space is quite foreign to most computing paradigms and represents a considerable deviation in how to think about programming. This also represents a real constraint on what NMC systems can do. Unlike conventional parallel systems, where scaling is associated with running a problem faster, it can be argued that bigger NMC systems are fundamentally better because smaller systems simply cannot implement certain models. This is increasingly important as the recent trends of neural network scaling are not promising for NMC, at least with respect to the algorithms being explored today. On conventional hardware, a larger ANN model simply requires more memory, whereas a bigger model on NMC requires an entirely bigger system.

Of course, this physical realization comes with a benefit as well—the program does not need to be retrieved from memory. This immediately cuts down on a substantial part of the cost of a serial program. From a time and energy perspective, the primary cost in NMC is in loading the model initially. As such, there is a rather significant advantage for identifying calculations that can largely reuse a set of predefined calculations, which allows the initial cost of loading the model to be amortized over many steps of the program. In neural algorithms, this re-use arises in feedback connections between neurons, referred to as *recurrence*. Such recurrence is a defining feature of biological neural circuits in the brain, and it has been observed that recurrent neural networks are preferential on NMC [20].

Notably, this use of recurrence is a clear distinction from how many algorithms are typically defined as well as specialized architectures like dataflow approaches. Most dataflow implementations are defined as acyclic graphs. Even in neural networks, where recurrent layers are often used, recurrence is typically local and 'unrolled' for purposes of deploying on many conventional architectures.

B. Neuromorphic computing is intrinsically parallel and asynchronous

Inherent in the graph-based description of NMC programs described above is the implication that NMC is intrinsically a parallel architecture. While details differ across NMC platforms, algorithmically NMC is typically be viewed as having every neuron acting independently and asynchronously from one another. It should be emphasized how different this extremely parallel neural programming paradigm is from conventional parallel architectures. The underlying serial nature of

[2]For intuition, consider a compute graph $G_{sigmoid}$ of sigmoid activated neurons; an equivalent spiking neural network ($G_{spiking}$) and an equivalent rectified linear neuron network (G_{ReLU}) can be constructed, but these computational graphs will be different.

Von Neumann is fundamental in most parallel architectures—adding more cores to a computation typically amounts to distributing a set of primarily serial workloads across processors and carefully coordinating the communication of information to align those workloads.

Parallel computing introduces a whole set of unique challenges at both the algorithm and architectural level, and one significant simplification that has led to significant efficiencies is to focus on "single-instruction, multiple data", or SIMD, approaches that shape algorithms to target a number of very powerful compute cores with a set of common instructions that are applied to a large volume of data. For this reason, many parallel architectures, such as GPUs, are SIMD. SIMD-based architectures clearly are powerful with appropriate algorithms, such as linear algebra applications where the required mathematical operations and memory accesses are highly structured. The downside of SIMD is that if an application lacks structure, such as Monte Carlo simulations with highly divergent trajectories, the uniformity of SIMD becomes a drawback. It is in this heterogeneous setting, which can be termed multiple instruction, multiple data (MIMD), that we suggest that NMC may thrive. NMC effectively provides a unique path to an MIMD architecture—since every calculation is implemented in its own population of neurons, there is no reason that these calculations need to be the same. In effect, if a complex diverse computation is going to be spread across a large population of neurons, there is no reason not to make the neural implementation match that diversity and complexity.

The flip side of spreading out a computation in this manner is that communication costs, yet again, can become problematic. The brain's solution to this, which has been adopted in most NMC systems, is to use *spiking* to minimize the costs of communication. Spiking refers to the event-driven communication of neurons, whereby a neuron only communicates if its inputs satisfy some condition, such as crossing a threshold. In the brain, this is an all-or-none single-bit—transmit a `1` or no transmission at all. In today's NMC platforms the spike may consist of more information, but they are always event-driven (see section III-C) and almost always target-agnostic (i.e., all targets receive the same spike). This creates a data-dependent cost that is not present in conventional computation. If `0`s are effectively free in communication and computation, sparsity becomes a particularly important feature of algorithm design.

An additional element introduced by the event-driven nature of spiking and the extreme parallelism of neural circuits is that time becomes important. Time is not really present in a serial Von Neumann setting beyond the ordering of instructions, and more often than not time is a challenge to be overcome in parallel computing through barriers and blocking communication. Rarely is time used as part of the computation. In contrast, in the brain *when* something happens is often as important as *what* happens. Increasingly, NMC algorithms have been shown to be able to use timing to perform computations efficiently, particularly in the implementation of graph algorithms [12], [21], [22].

The MIMD–SIMD difference of neuromorphic not only distinguishes it from GPUs, but it similarly separates it from many alternative in-memory computing architectures, such as resistive memory crossbars and optical computing approaches. Many in-memory computing approaches realize benefits not only from integrating computation with memory, but also by aligning the physics of hardware with the desired computation. While some systems preserve some level of programmability, most of these architectures generally can be viewed as SIMD (Figure 2). Because these architectures are non-Von Neumann and emphasize local memory, they are also often referred to as neuromorphic, but it is critical to appreciate the distinction of these systems in terms of programmability.

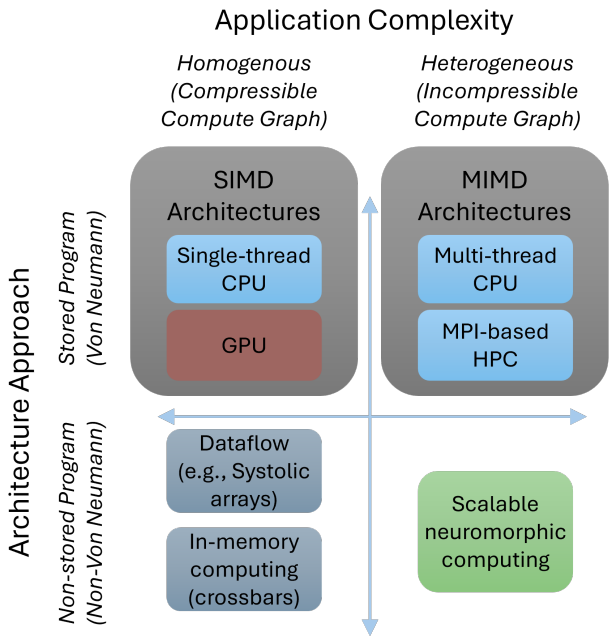

Fig. 2. Notional landscape of parallel computing approaches. For applications that are relatively homogeneous in their computational graph (e.g., the structured linear algebra of conventional neural networks), SIMD and dataflow architectures are effective, whereas for heterogeneous applications an MIMD or neuromorphic approach should be more effective.

C. Neuromorphic processing is data-dependent and often non-deterministic

The final distinction that we will discuss here is the unpredictability of neural computation. Biological neural systems are often considered noisy when compared to engineered systems. The extent and implications of this noise as it relates to the brain is debated; biological systems certainly do not have the same level of precision as digital systems, however it is not clear that their activity is impacted by biophysical noise. If anything, the presence of noise in brains appears to be intentional; for instance, the primary source of noise in brains is stochastic synaptic vesicle release, which is a biophysical process whose probabilities are tightly controlled and appear to be genetically linked to different neuron types. Stated differently, the noise in real neural systems is likely a feature and not a limitation.

Because communication is event-driven and neurons behave asynchronously, computation effectively becomes event-driven as well. An individual neuron processes information as it arrives. The combined effect of intrinsic stochasticity, distributed computation, and event-driven communication means that neuromorphic computation can easily become non-deterministic and non-repetitive. Stated differently, the same computation may be performed by many different patterns of neural behavior. How ensembles of such neurons can be constructed to produce reliable and arbitrary computations is an open question in the theoretical neuroscience field [23]–[25]. Nevertheless, it is widely recognized that equivalent biological computations are considered stable across a diverse set of neuron circuits [26], and thus computation abstracted to a higher level than neurons [24] can presumably be reliable even in a formal sense, just as sampling algorithms can be proven to approximate fixed distributions [27].

For this reason, many neural algorithms to date have minimized the use of stochasticity and asynchronous communication in their design or have included it in a very deliberate manner [28], [29]. There are reasons to believe that ubiquitous stochasticity could provide fundamental improvements to algorithms [30], and it is very likely that the ability of neuromorphic to enable sampling alongside of memory and compute provides added computational value.

Similarly, learning represents a form of non-determinism in an NMC system that is challenging to evaluate rigorously. In a sense, learning is a property of an algorithm, and the ability of an architecture to permit learning simply broadens the class of suitable algorithms. At the same time, the ability to learn probably does yield complexity advantages in that an algorithms initial implementation costs can be amortized over a longer lifetime in the face of context drift [31].

While the non-determinism and adaptability of biological neural circuits makes neural computation richer and potentially more computationally powerful from a theoretical sense, this complexity also risks making NMC more challenging for those familiar with precise and sequential deterministic algorithms. For this reason, and given the relatively immature state of probabilistic and adaptive neural algorithms, we will not examine these architectural benefits further here but simply acknowledge them as fundamental differences between NMC and conventional approaches. As the next sections describe, the differences between NMC and conventional architectures can be analyzed at a more fundamental level and future work will demonstrate how these distinctions can amplify the effects of ubiquitous stochasticity and learning.

IV. What neuromorphic is good at

From the above description, we can see that NMC is an non-stored program, MIMD architecture where the computation is fully distributed over processing elements. In this sense, it is similar to a distributed memory MIMD architecture. However unlike standard MPI-based distributed computing approaches, where each core is responsible for executing a thread with a communication strategy such as MPI responsible for aligning spatially disparate threads, neuromorphic algorithms must realize the full algorithm spatially and asynchronously. The question that presents itself is whether there is any fundamental algorithmic advantage to using such an architecture?

Here, we will consider three primary theoretical metrics: how an NMC system impacts the time (T), space (C), and energy (E) of an algorithm. Most models of parallel computation consider the implications of parallel processors sharing a memory (e.g., the PRAM model [32], [33]), but there is no memory access in a neuromorphic architecture. However, from a processor perspective many of the same frameworks apply. For instance, it is useful here to consider a neuromorphic algorithm fully expanded out as a directed acyclic graph (DAG), which we call G per definition II.4. Summarized results of complexity scaling, which will be derived in the following sections, are presented in Table II.

TABLE II
Time, Space, and Energy Scaling of Neuromorphic and Conventional Systems

	Time (T)	Space (S)	Energy (E)
Conventional			
Ideal	$O\left(\frac{T_1}{p}\right)$	$O(p)$	$O(T_1)$
CPU (Realized)	$O(T_1)$	$O(1)$	$O(T_1)$
GPU (Realized)	$O\left(\frac{T_1}{p \times p_{efficiency}}\right)$	$O(p)$	$O(T_1)$
Neuromorphic			
Ideal	$O(T_{\inf})$	$O(T_1)$	$O(\Delta G)$
Realized	$O(N_{core} T_{\inf})$	$O(T_1/N_{core})$	$O(\Delta G)$

For a computational graph, G: T_1 describes the total work (also the time on a serial processor), $T_{\inf}$ describes the minimal depth, and ΔG describes how information changes within G in the course of a computation. p describes the number of distinct conventional cores or threads, $p_{efficiency}$ describes the ability to use parallel threads in a SIMD framework, N_{core} describes the number of neurons per neuromorphic core.

A. Parallel analysis of neuromorphic algorithms

1) Time: Often workloads on a parallel system are analyzed from the perspective of Amdahl's Law [34], however due to our algorithm graph formulation, here we leverage instead Brent's Theorem [35] which provides insights into parallel algorithm design. Stated differently, Amdahl's law is suitable when the algorithm is fixed and the parallel system is a free variable, but in our case we explore when the algorithm (whether conventional or neuromorphic) is a free variable. Brent's Theorem (which in its naive form is agnostic to hardware) can be stated as

$$\max\left(T_{\inf}, \frac{T_1}{p}\right) \leq T_p \leq \frac{T_1}{p} + T_{\inf} \tag{1}$$

where T_1 is the total number of compute elements in G (i.e., the total number of operations that need to be performed, and thus the time to compute the program serially), p is the number of processing elements, $T_{\inf}$ is the minimal depth of the DAG (i.e., the minimum amount of time it will take to implement the program), and T_p is the time for the system itself to implement a conventional program. [3]

Brent's theorem can inform us directly about the value of NMC parallelism. Since NMC is not stored-program, all steps of the program *must* be fully realized by some dedicated computing resource (e.g., a subset of neurons) that can perform each operation in G. Through this logic, if an NMC is implementing G, we can assert that $p \geq T_1$. Restating Brent's Theorem for the time for a neuromorphic solution, T_N, we get

$$T_N = T_{\inf} \tag{2}$$

In some respects, equation 2 is trivial, it simply states that with total parallelization, the time of computation is bound by the circuit depth. However, it is useful to state explicitly, because this is how NMC operates—the number of processors is not a free variable as in conventional parallel systems, rather it is derived from the algorithm graph G.

The above formulation is useful to set a bound on the time of a NMC computation. Each operation in G may require a population of neurons configured in a graph itself to perform that operation. However, so long as this neural compute graph for each operation in G is constant-depth with respect to the fan-in (which is likely for any algorithm defined for conventional operations), an ideal NMC program should operate at a speed defined by the circuit depth $O(T_{\inf})$.

2) Space: However, time is only part of the story; space and energy are equally important resources to consider as well. The most clear-cut difference between a stored-program architecture and a neuromorphic architecture is the physical space required for the computation. Space is critical not only for SWAP (size, weight and power)-constrained systems, but also for forecasting the energy costs of data movement.

The space required in a conventional architecture, C_p, is relatively trivial; it effectively is a constant cost for each of the processors and the size of the memory required for the program itself plus any intermediate states required.

$$C_p \geq c_p p + C_{program} + C_{data} \tag{3}$$

The processor space cost ($c_p p$) is straightforward; each processor takes a certain amount of physical space on a chip, c_p, and there are p processors. The memory cost is more complex. At minimum, it is the bounded by the size of the program itself and the data that the program uses and manipulates in the course of its operation. This is a lower bound, in conventional systems it is often useful to store the program and relevant data redundantly within memory local to each processor to minimize data communication.

NMC space costs are considerably different. Because processing elements are not dynamically repurposed in neuromorphic systems according to a stored program, at worse case the required size of a neuromorphic system will scale with the overall size of the algorithm graph, G itself. As stated above, the depth of G is $T_{\inf}$ and that the total number of operations is T_1. On each level $i \in T_{\inf}$ of G, there are m^i operations, such that $\sum_i^{T_{\inf}}(m^i) = T_1$.

Complicating the NMC analysis is the fact that multiple neurons may be required to implement each operation in G, a number we denote as N_p. Further, these operations need not be identical (as in a feed-forward neural network) and thus can be heterogeneous in their neuron requirements; so we can accordingly index the requirement for each operation $j \in T_1$ of G as N_p^j neurons required. From the compute graph, we can then count the total number of require neurons, N_{total} as

$$N_{total} = \Sigma_{j=1}^{T_1} N_p^j \tag{4}$$

The total synapse count, S_{total} can be similarly constructed from the compute graph once expanded to neurons.

If c_N is taken as the hardware footprint of an individual neuron and c_S is the hardware footprint of a synapse, then the total space cost of an NMC implementation will be bounded as

$$C_N \approx c_N N_{total} + c_S S_{total} \tag{5}$$

From the above analysis, it can be seen that while NMC time is proportional to the depth of G, $T_{\inf}$ (Equation 2), the spatial cost of the implementation (Equations 4-5) is proportional to the total operations in G, T_1. This trade-off is effectively the inverse of conventional computing, suggesting there is no intrinsic time advantage of NMC parallelism. Therefore, achieving a time–space NMC advantage over conventional hardware requires that a NMC solution preferentially enables efficiencies in the computational graph itself.

One such method to compress the computational graph in time. For space efficiency, it is desirable that elements can be reused if the same computation is going to be repeated over and over again. We have to be careful here, as reusing processors in a program-dependent manner is effectively the stored-program architecture. So for the purposes of this analysis, we consider specifically the special case where only full levels of G are reused and the exit from recurrence is limited to only one branch. Importantly, neither of these assumptions are critical; indeed the brain can be viewed as having numerous exceptions to both , and in practice NMC algorithms will benefit similarly from more complex strategies.

Similar to the case of perfect parallelism in equation 1, the ability to reuse neurons across layers of G is similarly bounded by the computational depth of G, that is $T_{\inf}$. This introduces

[3]For intuition, consider the equation $y = ((5-2) \times (3-1))^3$. The DAG for this equation's computational graph has three levels: subtraction, multiplication, and exponentiation, and thus the DAG has a minimal depth $T_{\inf} = 3$ and a total number of operations $T_1 = 4$ or $T_1 = 5$ depending on whether the exponent is applied to the original factors or the final product. Brent's theorem states that even with infinite processors, you still must wait at least 3 operations to be performed.

a lower bound as a best-case scenario in terms of space is the condition of a fully recurrent circuit, whereby each level in G is identical to all of the others, that is $\forall i, j \in T_{\text{inf}}, m_i = m_j$. The upper bound is the worse-case where every operation has to be separately realized in hardware.

Based on this, we can extend equation 5. If we assume that $\bar{N}_p$ and $\bar{S}_p$ is the average neuron and synapse cost of operations in G, we can claim that

$$(c_N \bar{N}_p + c_S \bar{S}_p)\frac{T_1}{T_{\text{inf}}} \leq C_N \leq (c_N \bar{N}_p + c_S \bar{S}_p)T_1, \quad (6)$$

Importantly, the temporal compression seen due to recurrence does not speed up a NMC solution, but it does lower the overall spatial cost in a manner that is not immediately available for optimizing a conventional stored-program architecture (as those processors are reused already).

3) Energy: From equations 2 and 6, it is clear that for a NMC algorithm is ideal from a time perspective to minimize the depth T_{inf} and from a space perspective to maximize the recurrence. However, these optimizations do not directly account for the third resource, energy, which we will see is highly dependent on the nature of the computation itself.

Energy is often cited as an advantage of NMC for multiple reasons, but only some of these several reasons point to a scalable algorithmic energy advantage. One such advantage is an implication of being an in-memory architecture. Due to the in-memory nature, state does not need need to reset regularly to allow reuse of shared logic units, but rather state only changes when the computation requires it to do so. This allows event-driven updates to be used at the architecture level, which in turn makes energy proportional to the change of state across the computational graph.

When we are computing the energy cost of an algorithm, ultimately what matters boils down to change of state——flipping a 1 or 0 in a digital sense, either in a register, a logic circuit, or a wire (for communication). Each of those have different scaling factors, but they're all a change in state and they call contribute to energy in an electronic system.

Energy in conventional systems is proportional to total work. In conventional processing, we basically can assume that everything touched by an operation is changing equivalently no matter what the computation is. For example, because every line of code is agnostic to what came before it while using the same compute resources, things like the registers are constantly being overwritten and so we have to assume every write operation has to set every bit independently of what it last was. So each op has a certain cost no matter what the actual computation was. Memory access, communication, logic, etc all combines into a pretty easy estimation.

$$E_p = E_{operations} + E_{comm} \quad (7)$$

$E_{operations}$ is proportional to the total number of operations in the program, which we have already seen as T_1. We expect that the use of neurons compared to digital logic will impose different energy costs, especially if the computation differs due to an analog or digital implementation, but this will largely be a constant factor which we call e_{op}. [4]

$$E_{operations} = e_{op} T_1 \quad (8)$$

Communication, which in practice tends to dominate energy costs, is more complex. In a stored-program architecture that separates memory from processing, we expect multiple memory read and writes per operation, which results in significant data movement per operation. While the cost of a memory access is highly impacted by the memory architecture (SRAM vs DRAM, level of cache, etc), for this purpose we compress all the memory-related communication for an operation as e_{mem} Joules-per-bit, with a scaling factor of b_p bits-per-operation. Importantly, we expect that $e_{mem} >> e_{op}$, but again we approximate that this cost scales linearly with total program work.

$$E_{comm} = e_{mem} b_p T_1 \quad (9)$$

Energy in neuromorphic systems is proportional to the cumulative change of state. Energy must be estimated differently in NMC. Because processing elements are not shared, all changes in the electronics can be event-driven, potentially greatly reducing computational overhead. This makes energy hard to compute on average—simply adding up the operations no longer works because if the operation did not result in a change, then no energy is needed (i.e., if there is no spike and no neuron update, there is no energy expended). What matters in a neuromorphic system is what the actual change in the compute graph is.

In one respect, algorithm work is analogous to physical work, whereby work is the integral of force over a distance ($W = \int F \cdot d\sigma$). In computation, this change of distance ($d\sigma$) can be viewed as the **change of state**, which for conventional operations is typically constant (i.e., expected switches from `0` to `1`). This makes the total physical work, i.e., the energy, proportional to the sum of operations. In NMC, because operations are event-driven and bits are only changed if necessary, this change of state is *not* uniform. So for NMC, physical work should no longer be seen as proportional to the number of operations, but rather the cumulative change of state induced by those operations.

For our neuromorphic program that consists of a graph $G_N = (N, S)$ of neurons (N) and synapses (S), we will consider Δ as an operator that describes the change in a set of variables represented in hardware. The energy E_t at a given timestep, t, is equal to some combination of how the neurons and the synapses changed at that timestep, ΔN and ΔS.

$$E_t \propto \Delta\text{State} = e_{neuron}\Delta N + e_{spike}\Delta S \quad (10)$$

Importantly, this is where the analog versus digital difference can be factored in. In an analog system, the energy cost of an update is proportional to the update itself: a small

[4]Strictly speaking, we should expect a different e_{op} for every different operation, but the net effect should still be a constant factor.

change in a neuron's state variable (say -64mV to -63mV) is a tiny cost; but in a digital system that can be a huge change (i.e., 7 bits have to change state in converting `00111111` to `0100000`). Stated differently, ΔN is architecture and representation dependent and e_{neuron} is hardware dependent.

In a moderately active network, it is reasonable that all neurons will be receiving synaptic inputs ($e_{synapse}$), thus having a baseline level of internal state update ($e_{voltage}$), but only a subset of those neurons will fire ($e_{spikegen}$). If we consider f_t as the average firing rate of an individual neuron over some time-window, δt, and N_{total} as the total number of neurons, then we can break down

$$\begin{aligned} e_{neuron}\Delta N \approx & e_{voltage}N_{total} + \\ & e_{spikegen}f_t N_{total} + \\ & e_{synapse}f_t S_{total} \end{aligned} \tag{11}$$

The magnitude of each of these scalars ($e_{voltage}, e_{spikegen}, e_{synapse}$), is hardware dependent, and all are targets of materials and analog device research. Table III summarizes these constant factors. While the energy costs of synaptic operations has been explored in some NMC systems such as IBM TrueNorth [36], [37] and Intel Loihi [38], the relative importance of each of these constant factors is generally not measured directly.

Likewise, ΔS and e_{spike} are architecture and hardware dependent as well. If we are considering spiking systems specifically, then each spike represents a transient deviation from `0` to `1` and back to `0`. So effectively the communication cost of a neuromorphic system will be proportional to the number of spikes, $\Delta S \approx f_t N_{total}$.

There is an architecture-specific consideration as well; the cost of communication depends on the distance information must travel, which we refer to as ℓ (specifically, ℓ is the average distance that a spike must travel in an algorithm). In practice, much of algorithm design and the embedding onto a hardware platform focuses on optimizing placement to minimize ℓ, but here we simply acknowledge that it is a major consideration.

$$e_{spike}\Delta S \approx e_{spike}\ell f_t S_{total} \tag{12}$$

Together, this suggests the energy of a neuromorphic system can be estimated as

$$\begin{aligned} E_t \approx & e_{voltage}N_{total} + \\ & e_{spikegen}f_t N_{total} + \\ & e_{synapse}f_t S_{total} + \\ & e_{spike}\ell f_t S_{total} \end{aligned} \tag{13}$$

Per equation 13, we can see that energy scales with number of nodes and edges in G but in a less obvious manner. The first thing to note is that while synapses will always outnumber neurons, it will generally be the case that ℓ will also scale with N_{total} in some way, so $N_{total} < S_{total} < \ell S_{total}$. Thus if we ignore the scale of the constant factors, the communication of spikes across a NMC system will dominate the energy cost; though it is important to note that this is also the cost that can be most improved by algorithm design and proper embedding.

The second thing to note about equation 13 is that only the first term is not dependent on f_t. The generation, transmission, and reception of spikes should be expected to dominate the cost of any NMC algorithm. This is important to note when it comes to targeting hardware optimizations; we should expect that hardware optimizations targeting subthreshold neuron updates will have minimal impact to the system. The flip side is that using more sophisticated neuron models should not contribute significantly to the overall energy scaling. If a more sophisticated neuron model allows fewer neurons (and thus fewer synapses) to be used, the effects may be significant.

The third thing of note is that the energy described in equation 13 is per timestep. Activity (f_t) will change over the course of a neuromorphic solution. This is a consideration in edge applications where there is a maximum power (effectively the peak E_t) threshold. However, for most applications the energy of the whole calculation matters; which relates to the total time.

$$E_N = \sum_{t=1}^{T_N} E_t \tag{14}$$

As such, algorithm modifications that lower the number of timesteps (depth of circuit) may result in lower or higher overall energy consumption depending on how the overall firing rate or the number of neurons and synapses change.

V. Identifying a Neuromorphic Advantage

The previous sections outline how the time, space, and energy scaling of NMC is fundamentally different than conventional architectures. The costs of NMC are related to the number of neurons and synapses in the neural algorithm, but largely in an indirect fashion as compared to conventional architectures, whose costs typically scale approximately linearly with the computational work of an algorithm. This makes it challenging to develop clear apples–to–apples comparisons. However, we can make the following immediate observations from the cost factors described in equations 1 to 14.

- Time: Per equation 2, an ideal NMC architecture can implement any conventional algorithm at the minimal computational depth, T_{inf}. As T_{inf} is the lower bound for conventional architectures, ideal NMC should be faster (at worse the same speed) than conventional architectures from a time complexity perspective.
- Space: Per equation 6, the space cost of NMC architectures will scale with the overall size of the algorithm, whereas that is not necessarily the case for conventional architectures.
- Energy: The energy of a conventional algorithm (equation 7) effectively scales linearly with computational work of an algorithm, whereas the energy of an NMC algorithm (equation 10) effectively scales with the derivative of algorithm states. For this reason, at minimum an NMC algorithm could scale with the computational work (every

TABLE III
CONSTANT FACTORS IMPACTING NEUROMORPHIC COMPUTING

Constant factor	Description (Energy cost to...)	Hardware Equivalent	Digital Cost	Path to Improvement
$e_{voltage}$	Update internal neuron dynamics	Resistors, capacitors	Low	Analog CMOS
$e_{spikegen}$	Implement neuron non-linearity / spike generation	Analog-to-Digital Converter	Moderate	Analog CMOS
$e_{synapse}$	Deliver synaptic weights (may be stochastic)	Resistors, RNG	Low to medium	Memristors, TRNGs
e_{spike}	Communicate spikes between neurons	Information Routing	High ($\propto$ distance)	Optical, 3D

time-step represents a change from zero), suggesting conventional architectures provide an upper complexity bound for NMC for energy efficiency.

Combined, the differences in how costs scale between NMC and conventional are probably what we should expect. An ideal NMC architecture should be incredibly fast and energy-efficient, especially if the algorithm is well-suited for the architecture, but spatially expensive. This is consistent with the human brain itself. The brain performs complex calculations at extremely low computational depths (a batter can decide to hit a 100 mph pitch at the equivalent of 10 clock-cycles) with at exceptionally low power cost (estimated at lower than 20 W). However, this time and energy efficiency has a large spatial cost, requiring over 10^{11} neurons and 10^{15} synapses at levels of complexity far beyond any artificial neurons and synapses.

Given these relative costs, we can return to our original motivating question of what classes of algorithms should be well suited for NMC. At first glance, it is evident that algorithms with the following properties may stand to benefit:

- Algorithms where increasing the complexity of neurons and/or synapses could result in a corresponding decrease in number of neurons or synapses or the average activity of neurons. Examples may include neurons with multiple states, spatially distributed dendrites, and stochastic synapses.
- Recursive algorithms for which the algorithm graph G can be described with cycles allowing the same neuron resources can be reused. This lowers the overall spatial cost of NMC and decreases the distance penalty in communication.
- Iterative algorithms which are both recursive in nature and show a decreasing computational gradient over time as the algorithm approaches a solution. Decreasing ΔN and ΔS will directly result in a decreased energy cost.
- Probabilistic algorithms which require the same computational graph be sampled many times. If sampling is intrinsic to the implementation, in addition to effectively decreasing the size the ability to reuse of the same graph allows amortization of any constant cost.

A. A positive example: iterative mesh-based algorithms

To further explore this, let us consider a conceptual model of an algorithm that tracks the behavior of the system over M_S state locations. For example, in a finite element simulation, the state of all M_S locations are updated in an iteration, and in a discrete-time Markov Chain (DTMC) simulation M_S reflects the dimensionality of the transition matrix. Consider specifically the case where each mesh point only communicates at most with a finite number (scale-free) of other locations, which is defined as the mesh maximum fan-out, K. The model runs for M_T timesteps.

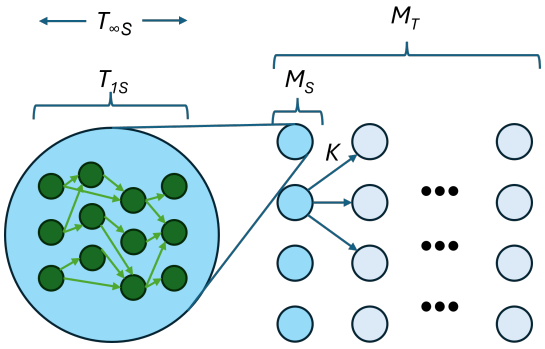

Fig. 3. Expanded graph of generic mesh problem

While it would never fully be fully expanded, the computational graph of this problem, G_{mesh}, would in fact be a hierarchical graph of graphs consisting of what computations must be performed at each mesh point. We consider that within each model timestep each mesh node consists of a computational graph of T_{1S} computations, consisting of a minimal depth of $T_{\inf S}$ to update the state of that mesh point. As there are M_S meshpoints and since the model consists of M_T timesteps, the fully expanded graph will be replicated $M_S \times M_T$ times.

Prior to any recognition of recurrence or parallelizability, we can quantify the size of G_{mesh} as $T_{1,mesh} = M_S M_T T_{1S}$ and $T_{\inf,mesh} = M_T T_{\inf S}$.

1) Conventional Implementation: In an NMC system, the description of the model at each state and the state itself (if being computed) must be physically realized in the neural fabric, so $C_p \approx M_S$. In contrast, in Von Neumann, the description of the model and its states will exist in memory. For non-trivial model sizes, this would primarily consist of non-local DRAM storage.

We can look to G_{mesh} to estimate the total computational cost of simulating our mesh on p processors.

Time: Per equation 1, the speed of a mesh-based algorithm on a parallel computer with p processors will be limited by the computational depth of the overall graph and the number of iterations, which is $T_{\inf S} \times M_T$.

Space: As with NMC, there is no need to store the entirety of G_{mesh} on the conventional hardware. Rather, we can assume an ideal conventional system with a common memory shared by all p processors. The size of that memory would scale with the number of mesh points, M_S

$$C_{p,mesh} = c_p p + c_{mem} M_S \tag{15}$$

Energy: Per equation 7, the energy of a conventional system will generally always scale with the total amount of work performed in the application. In this case, this effectively becomes the size of the fully unrolled G.

$$E_{p,mesh} = e_{op} T_{1,mesh} T_{1,mesh} = M_S M_T T_{1S} \tag{16}$$

2) Neuromorphic Implementation: Time: Based on equation 2, we can assert that the runtime of an ideal NMC implementation of the mesh will scale with $T_{\inf,mesh}$/

Space: We define N_{mesh} as the number of neurons per meshpoint and we recognize that $N_{mesh} \approx O(T_{1S})$. We know from the problem definition that the timesteps of G_{mesh} can be fully compressed into one recursive layer. As such, the total size of the NMC deployment will be

$$C_{N,mesh} = c_N N_{mesh} M_S \approx O(T_{1S} M_S) \tag{17}$$

Energy: Based on these assumptions, the NMC implementation of our application will require the following in terms of resources

$$N_{total} \approx M_S N_{mesh} \qquad \approx O(M_S) \tag{18}$$

$$S_{total} \approx M_S N_{mesh} K \qquad \approx O(M_S) \tag{19}$$

The spatial structure of mesh-based algorithms also allows us to further constrain the spatial cost of communication NMC. Specifically, because mesh points communicate locally, the size dependent term ℓ in equation 12 can be assumed to be constant ($\approx O(1)$). This allows the worst-case scaling of energy-per-timestep (equation 13) to be stated as

$$\begin{aligned} E_{t,mesh} &\approx (e_{voltage} + (e_{spikegen} + e_{synapse} + e_{spike}) f_t K) N_{total} \\ &\approx O(M_S) + f_t O(M_S) \\ &\approx O(M_S) \end{aligned} \tag{20}$$

Equation 20 highlights that, at worse, the energy consumption of an NMC mesh-based solution will scale linearly with the number of mesh points.

However, we can consider the case where our algorithm is an **iterative algorithm** such that the mesh points converge on a steady-state. As a network approaches steady-state, a ideally-designed NMC algorithm should similarly converge, leading to a decrease in f_t. While three terms in equation 13 scale directly with f_t, the first term does not. However, as $f_t \to 0$, we can interpret f_t as a probability that a neuron spikes in that time window. For the case where $0 \leq f_t << 1$, it becomes possible that some neurons do not receive additional inputs. As such, we can reinterpret the first term of equation 11 as $e_{voltage} N_{total}(1 - (1 - f_t)^K)$. While this term will approach zero slower than terms directly scaled by f_t, it demonstrates that for such an iterative algorithm $\Delta N \to 0$, and thus $E_t \to 0$.

B. A less-promising example: dense linear algebra-based ANNs

As a contrast, it is useful to consider a numerical computing example that is not inherently advantageous for neuromorphic. ANNs are typically defined in terms of linear algebra functions, such as vector, matrix, and tensor multiplications. This linear algebra formulation of ANNs motivated the initial use of GPUs for ANNs [39], [40] and ultimately the explosive development of large-scale linear algebra-based ANNs [14].

While this linear algebra approach is seen as a natural fit to crossbar-based neural accelerators [41], [42], promising results for scalable spiking neuromorphic platforms on ANNs have been limited to narrow cases [20]. The benefits of an event-driven neuromorphic system described above are less clear-cut.

To illustrate this, we consider the computational graph of an ANN, and since ANNs are typically a sequence of non-identical layers, for simplicity we focus on the graph of a single feed-forward layer, termed G_{ff}. A single layer of a neural network performs the following steps:

$$\begin{aligned} \mathbf{x}_j &= W \mathbf{y}_i \\ \mathbf{y}_j &= f(\mathbf{x}_j) \end{aligned} \tag{21}$$

whereby $f(\cdot)$ is a simple non-linearity, as in Table I.

This graph classically is described in two stages: a vector–matrix multiply followed by a non-linearity operated over a vector. However, for a computational implementations it is useful to view it as three distinct stages: a layer consisting of every synaptic operation ($s_{ij} = w_{ij} y i$), of which there are $S_{total} = N_i N_j$ operations, every neuron summation ($x_j = \Sigma_i s_{ij}$), of which there are N_j operations, and every neuron non-linearity ($y_j = f(x_j)$), of which there are also N_j operations. The minimal depth of this circuit is $T_{\inf} = 3$ and the total operation count, T_1 scales with the number of synapses, $O(N_i N_j)$.

1) Conventional implementation: Conventional implementations of equation 21 is straightforward, but nonetheless has been highly optimized over recent years due to the rise of ANN applications. As with the prior example, the time and space conventional costs are straightforward; but we note specifically the that the energy cost will scale quadratically with the size of the layers.

$$E_{p,ff} \approx O(N_i N_j) \tag{22}$$

2) Neuromorphic implementation: Time: As with the prior example, the NMC time cost of a feed-forward ANN layer is driven by the depth of the circuit, in this case $T_{\inf} \approx 3$.

However, unlike the mesh example, the operations being performed at each step of the graph are native to neurons, so

Space: Unlike the prior example which considered iterative algorithms on a fixed mesh, the dense structure of ANNs represents a problem for NMC implementations. Unlike the mesh example, the space required on a NMC system, based on equation 5, will be dominated by the number of synapses which scales quadratically with network layer size. While synapses may be relatively simple, since external memory is not available this space must be realized in full in hardware. Likewise, for feed-forward networks there is no structure in G_{ff} that allows the reuse of neurons and synapses.

Energy: The dense structure of feed-forward ANNs is similarly problematic for energy scaling on NMC as well. Because the number of synapses, synapse costs will dominate equation 13.

$$\begin{aligned} E_{t,ff} &\approx (e_{synapse} + e_{spike}) f_t S_{total} \\ &\approx O(N_i N_j) \end{aligned} \tag{23}$$

As with space, this quadratic scaling of energy is likely prohibitive for realizing an ANN advantage of NMC hardware. This scaling highlights that ANNs, as typically defined, are not well-suited to minimize (ΔState), as state changes continuously throughout the compute graph. **Thus, for ANNs, ultimately an ideal NMC system should scale similarly to conventional hardware in terms of energy, while residing on the same space / time trade-off.**

While this analysis shows that *direct* mappings of ANNs to NMC platforms should not be expected to provide scaling advantages, the prior analysis illustrates that advantages may be obtained if the computational graph of ANNs can be reshaped suitably. For this reason, research into SNN mappings likely should focus on identifying isomorphic computational graphs that can shift the neuromorphic costs. For instance, a NMC implementation of an ANN that is limited to a constant fan-in onto neurons or a scale-independent activation sparsity could shift the ANN costs closer to the mesh costs. This goal is similar to conventional methods focused on ANN quantization, pruning, and regularization to minimize conventional resource costs, which can be factored into ANN training methods, however the specific approaches would likely need to be specific to NMC.

An alternative is that NMC solutions should minimize the use of feed-forward layers and focus on recurrent networks. This is consistent with a growing belief in the NMC community that recurrent ANN models, such as echo-state networks [43], liquid-state machines [28], and more recently state-space models [44], [45], may be more suitable for NMC than feed-forward models. Recurrent networks naturally enable the reuse of components (mitigating the high space costs of NMC), however as described above, it is likely necessary to have either scale-free connectivity (constant fan-in on neurons) or a convergence of network activity to observe scalable energy advantages. To date it remains an open question whether such recurrent networks can be realized with these energy characteristics while providing competitive performance.

VI. From Ideal to Reality

The above analysis focuses on ideal implementations of both NMC and conventional computing. While this is reasonable for understanding the theoretical computing landscape, there are significant considerations when looking at the reality of how these parallel architectures are constructed.

A. Reality of neuromorphic

As described in section III-B, NMC is conceptually an MIMD architecture; similar conceptually to a massive multi-core CPU system. In practice, today's scalable NMC systems leverage a hierarchy of neuromorphic cores that each are responsible for a certain number of neurons and synapses and these cores generally are moderately programmable with local memory defining the neurons and synapse parameters. Nonetheless, within a core the simulation of neurons and synapses is somewhat serialized. If we consider a system that has n_{cores} separate neuromorphic cores, we can define N_{core} as the number of neurons per core, and S_{core} as the number of synapses per core. This changes our top speed of neuromorphic system from equation 2 to

$$T_{inf} \leq T_N \leq N_{core} T_{inf} \tag{24}$$

as every irreducible step through the model may require that all neurons on each core are cycled at least once. At the same time, the total space requirement for NMC will decrease by a comparable factor. Rather than the physical space being directly proportional to the size of the compute graph (equation 6), it can be scaled down since n_{core} neurons share the same physical compute structure.

$$c_N \frac{T_1}{T_{\text{inf}} N_{core}} \leq C_N \leq c_N T_1 / N_{core} \tag{25}$$

Finally, the energy of NMC in reality will be rather complex and depend on numerous factors, but we can observe that a physically smaller system should lower the spatial cost of spike communication, which is the largest contributor to energy in equation 13. Offsetting this benefit, however, is that spike transmission in real NMC systems is not point-to-point, but rather must go through some routing system. This routing will introduce a complex cost hierarchy and makes the cost of spike transmission highly dependent on the algorithm–hardware embedding.

B. Reality of parallel conventional systems

In contrast, the most widespread types of conventional architectures are multi-core CPUs, where the number of cores is typically $1 \leq p \leq 64$, and GPUs, where the number of concurrent threads can be in the thousands. The trade-off with GPUs is that these threads come at the expense of being SIMD, and accordingly, the scaling of GPUs the ability to relates directly to the structure of the compute graph. Specifically, GPUs are limited not by the number of processors, but rather

the number of effective threads that can be run concurrently. Therefore, while both multi-core CPUs and GPUs will scale with Brent's Law (equation 1), they do so with relation to the effective number of threads that can be used at a time, $p_{threads}$.

$$max(T_{\inf}, \frac{T_1}{p_{threads}}) \leq T_p \leq \frac{T_1}{p_{threads}} + T_{\inf} \quad (26)$$

For MIMD CPUs, $p_{threads} = p$, but for GPUs the calculation of effective threads is non-trivial. In effect, the ability to maximally utilize a SIMD GPU relates to the number of isomorphic subgraphs in G that can be identified. [5]

In practice this problem of partitioning a computational graph into the maximum number of isomorphic subgraphs is itself is an NP-hard problem, so effectively GPUs are best used in domains where the problem formulation itself presents a path to identifying those isomorphic components, such as in linear algebra or in code that lends itself to highly compressed descriptions.

Acknowledging that the analysis of G and identification of the number of usable threads is algorithm specific, we can still consider that any computational graph will have a thread efficiency, which we define as

$$p_{efficiency} = p_{threads}/p \quad (27)$$

Notably, while the speedup of parallelization in a SIMD system will scale with $p_{threads}$, the size of the conventional system will still scale with p even though moving to SIMD should represent a decrease in the constant factor with regard to size. Stated differently, GPUs represent a more compact form of parallelism than simply adding more cores to a CPU, but the trade-off is the SIMD constraint. Similarly, it should be expected that the overall energy consumption should not decrease significantly from the use of a SIMD architecture aside from the constant factor conferred by using a more compact architecture if the GPU can be leveraged at full potential.

C. Implications of the reality of neuromorphic and conventional

While the above sections could be extended in more depth, specifically to include the costs associated with a hierarchy of spike routing in NMC and differing costs of conventional memory access, they begin to illustrate where the key opportunities may reside for NMC versus conventional architectures.

Specifically, for tasks such as linear algebra which are highly homogeneous, we would expect (and indeed observe) a high $p_{efficiency}$ for SIMD architectures. While the theoretical scaling of GPUs would not be different from that of multi-core CPUs, their ability to fit 100x processing in a similar power and space budget provides a significant advantage in the constant factors associated with the scaling. In contrast, for more heterogeneous computational graphs, the advantage of NMC is evident. While the constant factors and size scaling penalties associated with NMC are significant, the potential that ideal parallelism can be achieved irregardless of G is attractive. The nature of G thus becomes critical. For SIMD architectures most algorithms can be formulated as linear algebra problems at some cost, presenting a challenge in graph homogenization to increase the $p_{efficiency}$ for a larger G_{SIMD}. In contrast, an NMC workload ideally should target the minimal G, as it is not penalized by heterogeneity. while a GPU approach should aim for a balance between an altered G_{SIMD} which trades off a larger size for more homogeneity, allowing the full potential of a GPU to be used, and a weaker utilization of its many threads with minimal change of the graph. Arguably this is what has happened with the Hardware Lottery; whereby AI algorithms are developed in a fully expanded form so as to be GPU compatible.

For this reason, the domain of an NMC advantage extends from computational problems that are compressible in time (allow recurrence and reuse of G components directly) to problems that are also incompressible in space (allow the relatively expensive MIMD nature of a NMC architecture to be maximally leveraged). This would extend the realm of potential NMC advantage from the iterative solvers and recurrent neural networks described earlier to also including tasks like large-scale graph traversal and agent-based simulations.

VII. Conclusions

A. Common thread of neuromorphic advantages

The above examples highlight several important points to consider in the context of the broader analysis described here.

First, there is likely no intrinsic *algorithm-independent* theoretical advantage of NMC. From a complexity perspective, the worst-case energy costs of NMC are equivalent to the expected costs of conventional and the time advantage of neuromorphic is directly related to the number of processors deployed (which in theory could be similarly scaled up for a conventional system). This should be expected—in a very real sense our framework for NMC is equivalent to an ideal parallel computer. That is not to say that there would not be an empirical advantage with scaling; we effectively are contrasting NMC to a PRAM model where all conventional processors have a shared memory that has a scale-free access cost. Nevertheless, as we are considering an ideal NMC system it is reasonable to overlook this limitation of parallel conventional systems.

Second, the illustration of iterative algorithms highlights that a significant theoretical benefit is possible if the task is suitable and the NMC algorithm is appropriately designed. Because energy in NMC scales with changes in state, an NMC algorithm tailored to similarly converge its representations

[5] We can decompose $G = (\mathcal{V}, \mathcal{E})$ into thread parallel subgraphs as follows: Suppose $H = (\mathcal{V}_H, \mathcal{E}_H)$ is a subgraph of G where $\mathcal{V}_H \subseteq \mathcal{V}$ and $\mathcal{E}_H \subseteq \mathcal{E}$ such that $\mathcal{E}_H$ connects the nodes in $\mathcal{V}_H$, and we can determine if two subgraphs H_1 and H_2 are isomorphic if they have a common relationship between their constituent nodes and edges. We consider that we can partition G into k families of isomorphic subgraphs $\{A_1, A_2, \ldots, A_k\}$ where each family A_i consists of a_i isomorphic subgraphs, and $G^{'}$ is the residual part of G that does not belong to any isomorphic subgraphs. In this case, the number of effective threads will relate to the distribution of A, such as $p_{threads} = max_{i=1\ldots k}(a_i)$.

($\Delta N \rightarrow 0, \Delta S \rightarrow 0$) as an iterative problem converges can see real benefits in total energy consumption. This not only suggests benefits for iterative numerical methods as suggested here, but has potential benefits for other tasks such as machine learning and optimization algorithms.

Third, this analysis demonstrates that NMC provides at worse comparable scaling to conventional when the computational graphs are equivalent, which is always an option for NMC. While not extensively explored, it has been shown that NMC architectures can enable strictly more efficient computational graphs [21]. For instance, even simple TG models can permit a logarithmic advantage in depth [46], which would be directly realized in this analysis as both a time and energy advantage. It is likely that incorporating more complexity into neuron and synapse dynamics can similarly enable the realization of more compact computational graphs. These benefits would build on the baseline advantages and characteristics of NMC scaling demonstrated here. For example, in [47], [48], the computational graph developed for Monte Carlo sampling of DTMC models leverages a distinct neuromorphic algorithm to solve the same math. Not only are the empirical advantages seen there due to the decay of activity ($\Delta N \rightarrow 0$) as the Monte Carlo simulation progresses, they also benefit from random number generation being similarly local to the processing—a feature that may be further be amplified with the incorporation of true random number capabilities in hardware [30].

Finally, this analysis has been to be fully agnostic to the underlying materials and device implementations of an NMC system. Moving from digital to analog or from silicon to non-silicon devices will have immediate consequences on many of the scaling factors in the above analysis, but this should not change the underlying architectural Big-O scaling. That being said, the analysis above targets an ideal NMC system, and the evolution of NMC towards this ideal parallelism and connectivity may require less conventional materials and devices than those used today. Additionally, moving away from digital devices will almost certainly allow the extension of the low-level capabilities of neurons and synapses, opening the door towards algorithmic advantages (i.e., more efficient G). Importantly, this analysis demonstrates the importance that any such exploration of advanced low-level capabilities be made in conjunction with architectural analysis such as this and co-design with algorithm development.

B. Limitations of this analysis

This analysis focuses primarily on the time, space, and energy complexity of ideal neuromorphic and conventional architectures. While we do briefly consider the implications of realistic implementations, specifically digital NMC systems that leverage neuromorphic cores contrasted to GPUs, this analysis could be extended considerably. Specifically, we only briefly discuss the costs associated with hierarchies of communication in neuromorphic cores and chips and conventional shared and local memories. Memory hierarchies have a significant effect on time and energy costs in conventional systems and thus are an area of considerable research today. Likewise, NMC hierarchies will similarly increase communication and impact time and energy costs for real NMC systems. While we do not expect the core complexity findings of this analysis will be impacted by these hierarchies, further analyses specific to each NMC system will be necessary to understand the points where these costs become prohibitive.

Additionally, this analysis does not consider precision of a computation. In digital systems, precision manifests itself directly in the computational graph G, however it can be expected to be an expansion of G that is compressible and thus easily handled within realized conventional systems if required precision is homogeneous. The impact of precision on NMC systems is an open question in NMC algorithm design and represents a potential cost that must be considered its suitability. Further, any move to leverage analog or non-conventional components in NMC systems must consider the impact on precision. Importantly, the precision demands of numerical algorithms are already being challenged in machine learning applications, and it is possible that the increased use of probabilistic approaches identified here may also mitigate some of the need for extreme high precision.

This analysis proposes a concrete hypothesis that the computational graph of an algorithm can be used to identify NMC suitability. This hypothesis merits a deeper investigation both from an empirical perspective—do the actual energy costs of today's NMC systems fit this framework?—and from a theoretical perspective—are the concepts of graph compressibility over space and time explored here suitable to characterize for classes of real applications? Empirical validation could involve benchmarking neuromorphic systems against GPUs for iterative optimization algorithms, measuring energy costs, and analyzing how sparsity impacts performance. The benchmarking of NMC has been a small but growing endeavor [49], [50], however these early efforts have been more application-driven as is typical in machine learning [51] and it remains an open question how to account for the fundamental architectural differences of NMC. Specifically, the breakdown of costs highlighted in Table III highlights how different hardware decisions can impact the relative costs of different algorithm components, greatly complicating applies-to-apples benchmarking.

Lastly, as mentioned in section III-C, we intentionally do not fully explore stochasticity and learning in this analysis. Ultimately, both stochasticity and learning can be viewed as algorithm design choices, but they may interact uniquely with the fully parallel and event-driven nature of NMC hardware, providing theoretical benefits far beyond what is identified here. For instance, when viewed in terms of the computational graphs analysis above, both stochasticity and learning could be viewed as mechanisms in which a NMC approach can uniquely compress the computational graph. This question of how this non-determinism of NMC algorithms relates to NMC architectures is a significant open question to explore going forward.

The addressing of these limitations is a significant challenge, but future research clarifying these questions will help refine a

theoretical framework for NMC that can provide a useful guide to identifying the true potential for neuromorphic approaches to computation.

C. A path forward for neuromorphic computing

Despite its recent successes in achieving brain-like scales [1], NMC still faces a number of challenges to become widely adopted as a complementary platform for numerical computing and AI. NMC systems remain difficult to program, with no easily accessible equivalent to PyTorch or a lower-level interface like CUDA. While there are emerging efforts to identify common programming frameworks [19], [52]–[54], the rapid growth of the field—both in terms of hardware and algorithms—makes it difficult for the broader community to converge on a common programming models that are robust to new hardware and applications yet make neural computation more broadly accessible.

Additionally, the diversity of NMC hardware looks likely to continue to grow. Due to the slowing of Moore's Law, conventional CMOS scaling looks unlikely to continue indefinitely, meaning that non-CMOS materials and alternative architectural approaches, including analog and optical, may be required to achieve human-brain scales at reasonable space and energy footprints. While the analysis here suggests that these approaches are unlikely to change the fundamental theoretical advantages of neuromorphic, they may make it more likely to realize the idealized benefits described here in future systems.

Despite these challenges going forward, the analysis in this paper demonstrates that there is a benefit to begin identifying algorithms that can benefit from NMC. Strengthening this fundamental understanding of what makes a strong NMC algorithm will be critical going forward to help overcome the future hardware and programming challenges mentioned above. The potential necessity for probability theory and graph analytics to influence this algorithm design places NMC research as much in the domain of computer science and applied mathematics as in its traditional fields of electrical engineering and physics. Additionally, from a neuroscience perspective it can be argued that we are at an early stage of leveraging brain-inspiration for computing and AI [55]. These opportunities suggest that through suitable algorithm design and focused brain-inspiration, NMC can help influence a revival of energy-efficient numerical computing and AI solutions.

Acknowledgements

The author thanks Darby Smith, Brad Theilman, and Craig Vineyard for comments and discussions. The DOE Advanced Simulation and Computing program provided support for this work. This article has been authored by an employee of National Technology & Engineering Solutions of Sandia, LLC under Contract No. DE-NA0003525 with the U.S. Department of Energy (DOE). The employee owns all right, title and interest in and to the article and is solely responsible for its contents. The United States Government retains and the publisher, by accepting the article for publication, acknowledges that the United States Government retains a non-exclusive, paid-up, irrevocable, world-wide license to publish or reproduce the published form of this article or allow others to do so, for United States Government purposes. The DOE will provide public access to these results of federally sponsored research in accordance with the DOE Public Access Plan https://www.energy.gov/downloads/doe-public-access-plan .

References

[1] D. Kudithipudi, C. Schuman, C. M. Vineyard, T. Pandit, C. Merkel, R. Kubendran, J. B. Aimone, G. Orchard, C. Mayr, R. Benosman *et al.*, "Neuromorphic computing at scale," *Nature*, vol. 637, no. 8047, pp. 801–812, 2025.

[2] P. A. Merolla, J. V. Arthur, R. Alvarez-Icaza, A. S. Cassidy, J. Sawada, F. Akopyan, B. L. Jackson, N. Imam, C. Guo, Y. Nakamura *et al.*, "A million spiking-neuron integrated circuit with a scalable communication network and interface," *Science*, vol. 345, no. 6197, pp. 668–673, 2014.

[3] S. Furber, "To build a brain," *IEEE spectrum*, vol. 49, no. 8, pp. 44–49, 2012.

[4] J. E. Hopcroft and J. D. Ullman, *Formal languages and their relation to automata*. Addison-Wesley Longman Publishing Co., Inc., 1969.

[5] K. Hornik, M. Stinchcombe, and H. White, "Multilayer feedforward networks are universal approximators," *Neural networks*, vol. 2, no. 5, pp. 359–366, 1989.

[6] D. H. Wolpert and W. G. Macready, "No free lunch theorems for optimization," *IEEE transactions on evolutionary computation*, vol. 1, no. 1, pp. 67–82, 1997.

[7] D. V. Christensen, R. Dittmann, B. Linares-Barranco, A. Sebastian, M. Le Gallo, A. Redaelli, S. Slesazeck, T. Mikolajick, S. Spiga, S. Menzel *et al.*, "2022 roadmap on neuromorphic computing and engineering," *Neuromorphic Computing and Engineering*, vol. 2, no. 2, p. 022501, 2022.

[8] D. Marković, A. Mizrahi, D. Querlioz, and J. Grollier, "Physics for neuromorphic computing," *Nature Reviews Physics*, vol. 2, no. 9, pp. 499–510, 2020.

[9] J. B. Aimone, "A roadmap for reaching the potential of brain-derived computing," *Advanced Intelligent Systems*, vol. 3, no. 1, p. 2000191, 2021.

[10] H. Jaeger, B. Noheda, and W. G. Van Der Wiel, "Toward a formal theory for computing machines made out of whatever physics offers," *Nature communications*, vol. 14, no. 1, p. 4911, 2023.

[11] J. Kwisthout and N. Donselaar, "On the computational power and complexity of spiking neural networks," in *Proceedings of the 2020 Annual Neuro-Inspired Computational Elements Workshop*, 2020, pp. 1–7.

[12] J. B. Aimone, Y. Ho, O. Parekh, C. A. Phillips, A. Pinar, W. Severa, and Y. Wang, "Provable advantages for graph algorithms in spiking neural networks," in *Proceedings of the 33rd ACM Symposium on Parallelism in Algorithms and Architectures*, 2021, pp. 35–47.

[13] J. B. Aimone and O. Parekh, "The brain's unique take on algorithms," *nature communications*, vol. 14, no. 1, p. 4910, 2023.

[14] S. Hooker, "The hardware lottery," *Communications of the ACM*, vol. 64, no. 12, pp. 58–65, 2021.

[15] M. Horowitz, "1.1 computing's energy problem (and what we can do about it)," in *2014 IEEE international solid-state circuits conference digest of technical papers (ISSCC)*. IEEE, 2014, pp. 10–14.

[16] J. L. Hennessy and D. A. Patterson, "A new golden age for computer architecture," *Communications of the ACM*, vol. 62, no. 2, pp. 48–60, 2019.

[17] J. B. Aimone, A. J. Hill, W. M. Severa, and C. M. Vineyard, "Spiking neural streaming binary arithmetic," in *2021 International Conference on Rebooting Computing (ICRC)*. IEEE, 2021, pp. 79–83.

[18] K. Hornik, "Approximation capabilities of multilayer feedforward networks," *Neural networks*, vol. 4, no. 2, pp. 251–257, 1991.

[19] J. B. Aimone, W. Severa, and C. M. Vineyard, "Composing neural algorithms with fugu," in *Proceedings of the International Conference on Neuromorphic Systems*, 2019, pp. 1–8.

[20] M. Davies, A. Wild, G. Orchard, Y. Sandamirskaya, G. A. F. Guerra, P. Joshi, P. Plank, and S. R. Risbud, "Advancing neuromorphic computing with loihi: A survey of results and outlook," *Proceedings of the IEEE*, vol. 109, no. 5, pp. 911–934, 2021.

[21] J. B. Aimone, Y. Ho, O. Parekh, C. A. Phillips, A. Pinar, W. Severa, and Y. Wang, "Provable neuromorphic advantages for computing shortest paths," in *Proceedings of the 32nd ACM Symposium on Parallelism in Algorithms and Architectures*, 2020, pp. 497–499.

[22] B. Kay, P. Date, and C. Schuman, "Neuromorphic graph algorithms: Extracting longest shortest paths and minimum spanning trees," in *Proceedings of the 2020 Annual Neuro-inspired Computational Elements Workshop*, 2020, pp. 1–6.

[23] R. Yuste, R. Cossart, and E. Yaksi, "Neuronal ensembles: Building blocks of neural circuits," *Neuron*, vol. 112, no. 6, pp. 875–892, 2024.

[24] B. H. Theilman, F. Wang, F. Rothganger, and J. B. Aimone, "Decomposing spiking neural networks with graphical neural activity threads," *arXiv preprint arXiv:2306.16684*, 2023.

[25] C. H. Papadimitriou, S. S. Vempala, D. Mitropolsky, M. Collins, and W. Maass, "Brain computation by assemblies of neurons," *Proceedings of the National Academy of Sciences*, vol. 117, no. 25, pp. 14 464–14 472, 2020.

[26] E. Marder, "Variability, compensation, and modulation in neurons and circuits," *Proceedings of the National Academy of Sciences*, vol. 108, no. supplement_3, pp. 15 542–15 548, 2011.

[27] N. Metropolis, A. W. Rosenbluth, M. N. Rosenbluth, A. H. Teller, and E. Teller, "Equation of state calculations by fast computing machines," *The journal of chemical physics*, vol. 21, no. 6, pp. 1087–1092, 1953.

[28] W. Maass, T. Natschläger, and H. Markram, "Real-time computing without stable states: A new framework for neural computation based on perturbations," *Neural computation*, vol. 14, no. 11, pp. 2531–2560, 2002.

[29] W. Maass, "Noise as a resource for computation and learning in networks of spiking neurons," *Proceedings of the IEEE*, vol. 102, no. 5, pp. 860–880, 2014.

[30] S. Misra, L. C. Bland, S. G. Cardwell, J. A. C. Incorvia, C. D. James, A. D. Kent, C. D. Schuman, J. D. Smith, and J. B. Aimone, "Probabilistic neural computing with stochastic devices," *Advanced Materials*, vol. 35, no. 37, p. 2204569, 2023.

[31] T. J. Draelos, N. E. Miner, C. C. Lamb, J. A. Cox, C. M. Vineyard, K. D. Carlson, W. M. Severa, C. D. James, and J. B. Aimone, "Neurogenesis deep learning: Extending deep networks to accommodate new classes," in *2017 international joint conference on neural networks (IJCNN)*. IEEE, 2017, pp. 526–533.

[32] S. Fortune and J. Wyllie, "Parallelism in random access machines," in *Proceedings of the tenth annual ACM symposium on Theory of computing*, 1978, pp. 114–118.

[33] P. B. Gibbons, "A more practical pram model," in *Proceedings of the first annual ACM symposium on Parallel algorithms and architectures*, 1989, pp. 158–168.

[34] G. M. Amdahl, "Validity of the single processor approach to achieving large scale computing capabilities," in *Proceedings of the April 18-20, 1967, spring joint computer conference*, 1967, pp. 483–485.

[35] R. P. Brent, "The parallel evaluation of general arithmetic expressions," *Journal of the ACM (JACM)*, vol. 21, no. 2, pp. 201–206, 1974.

[36] A. S. Cassidy, R. Alvarez-Icaza, F. Akopyan, J. Sawada, J. V. Arthur, P. A. Merolla, P. Datta, M. G. Tallada, B. Taba, A. Andreopoulos *et al.*, "Real-time scalable cortical computing at 46 giga-synaptic ops/watt with˜ 100× speedup in time-to-solution and˜ 100,000× reduction in energy-to-solution," in *SC'14: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, 2014, pp. 27–38.

[37] F. Akopyan, J. Sawada, A. Cassidy, R. Alvarez-Icaza, J. Arthur, P. Merolla, N. Imam, Y. Nakamura, P. Datta, G.-J. Nam *et al.*, "Truenorth: Design and tool flow of a 65 mw 1 million neuron programmable neurosynaptic chip," *IEEE transactions on computer-aided design of integrated circuits and systems*, vol. 34, no. 10, pp. 1537–1557, 2015.

[38] M. Davies, N. Srinivasa, T.-H. Lin, G. Chinya, Y. Cao, S. H. Choday, G. Dimou, P. Joshi, N. Imam, S. Jain *et al.*, "Loihi: A neuromorphic manycore processor with on-chip learning," *Ieee Micro*, vol. 38, no. 1, pp. 82–99, 2018.

[39] R. Raina, A. Madhavan, and A. Y. Ng, "Large-scale deep unsupervised learning using graphics processors," in *Proceedings of the 26th annual international conference on machine learning*, 2009, pp. 873–880.

[40] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "Imagenet classification with deep convolutional neural networks," *Advances in neural information processing systems*, vol. 25, 2012.

[41] J. J. Yang, D. B. Strukov, and D. R. Stewart, "Memristive devices for computing," *Nature nanotechnology*, vol. 8, no. 1, pp. 13–24, 2013.

[42] F. Cai, J. M. Correll, S. H. Lee, Y. Lim, V. Bothra, Z. Zhang, M. P. Flynn, and W. D. Lu, "A fully integrated reprogrammable memristor–cmos system for efficient multiply–accumulate operations," *Nature electronics*, vol. 2, no. 7, pp. 290–299, 2019.

[43] H. Jaeger, "The "echo state" approach to analysing and training recurrent neural networks-with an erratum note," *Bonn, Germany: German national research center for information technology gmd technical report*, vol. 148, no. 34, p. 13, 2001.

[44] A. Voelker, I. Kajić, and C. Eliasmith, "Legendre memory units: Continuous-time representation in recurrent neural networks," *Advances in neural information processing systems*, vol. 32, 2019.

[45] S. M. Meyer, P. Weidel, P. Plank, L. Campos-Macias, S. B. Shreshta, P. Stratmann, J. Timcheck, and M. Richter, "A diagonal structured state space model on loihi 2 for efficient streaming sequence processing," in *2025 Neuro Inspired Computational Elements (NICE)*. IEEE, 2025, pp. 1–9.

[46] O. Parekh, C. A. Phillips, C. D. James, and J. B. Aimone, "Constant-depth and subcubic-size threshold circuits for matrix multiplication," in *Proceedings of the 30th on Symposium on Parallelism in Algorithms and Architectures*, 2018, pp. 67–76.

[47] J. D. Smith, A. J. Hill, L. E. Reeder, B. C. Franke, R. B. Lehoucq, O. Parekh, W. Severa, and J. B. Aimone, "Neuromorphic scaling advantages for energy-efficient random walk computations," *Nature Electronics*, vol. 5, no. 2, pp. 102–112, 2022.

[48] J. D. Smith, W. Severa, A. J. Hill, L. Reeder, B. Franke, R. B. Lehoucq, O. D. Parekh, and J. B. Aimone, "Solving a steady-state pde using spiking networks and neuromorphic hardware," in *International Conference on neuromorphic systems 2020*, 2020, pp. 1–8.

[49] C. Vineyard, S. Cardwell, F. Chance, S. Musuvathy, F. Rothganger, W. Severa, J. Smith, C. Teeter, F. Wang, and J. Aimone, "Neural mini-apps as a tool for neuromorphic computing insight," in *Proceedings of the 2022 Annual Neuro-Inspired Computational Elements Conference*, 2022, pp. 40–49.

[50] J. Yik, K. Van den Berghe, D. den Blanken, Y. Bouhadjar, M. Fabre, P. Hueber, W. Ke, M. A. Khoei, D. Kleyko, N. Pacik-Nelson *et al.*, "The neurobench framework for benchmarking neuromorphic computing algorithms and systems," *Nature Communications*, vol. 16, no. 1, p. 1545, 2025.

[51] V. J. Reddi, C. Cheng, D. Kanter, P. Mattson, G. Schmuelling, C.-J. Wu, B. Anderson, M. Breughe, M. Charlebois, W. Chou *et al.*, "Mlperf inference benchmark," in *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 2020, pp. 446–459.

[52] T. Bekolay, J. Bergstra, E. Hunsberger, T. DeWolf, T. C. Stewart, D. Rasmussen, X. Choo, A. R. Voelker, and C. Eliasmith, "Nengo: a python tool for building large-scale functional brain models," *Frontiers in neuroinformatics*, vol. 7, p. 48, 2014.

[53] J. E. Pedersen, S. Abreu, M. Jobst, G. Lenz, V. Fra, F. C. Bauer, D. R. Muir, P. Zhou, B. Vogginger, K. Heckel *et al.*, "Neuromorphic intermediate representation: A unified instruction set for interoperable brain-inspired computing," *Nature Communications*, vol. 15, no. 1, p. 8122, 2024.

[54] J. K. Eshraghian, M. Ward, E. Neftci, X. Wang, G. Lenz, G. Dwivedi, M. Bennamoun, D. S. Jeong, and W. D. Lu, "Training spiking neural networks using lessons from deep learning," *Proceedings of the IEEE*, vol. 111, no. 9, pp. 1016–1054, 2023.

[55] J. B. Aimone, "Neural algorithms and computing beyond moore's law," *Communications of the ACM*, vol. 62, no. 4, pp. 110–110, 2019.